SE320 Software Verification & Validation

Week 1: Overview

Fall 2020

First…

- Lecture is recorded from Zoom
- On Zoom, you will not appear in lecture recordings if you don’t ask questions
    - But! I want you to ask questions!
    - More on that later

Why Are We Here?

- (in this class)
- Many/most of you will go on to write software professionally
- You will be expected to produce software that (mostly) works as expected
- How do you do that?
  - What is “expected”?
  - What is “works”?
  - What is “mostly”?

Introduction

- *Software Testing* is a critical element of developing quality software systems
- It is a systematic approach to judging quality and discovering bugs
- This course presents the theory *and* practice of software testing
- Topics covered include:
  - Black-box and white-box testing, and related test data generation techniques
  - Tools for software testing
  - Performance testing basics
  - Security testing basics
  - New: testing software for bias

Course Objectives

- Understand the concepts and theory related to software testing
- Understand the relationship between *black-box* and *white-box* testing, and know how to apply as appropriate
- Understand different testing techniques used for *developing test cases* and *evaluating test adequacy*
- Learn to use *automated testing tools* in order to measure *code coverage*

Course Trivia

- Instructor: Dr. Colin S. Gordon
- Email: csgordon@drexel.edu
- Office: My attic
- Office Hours:
  - TBD
  - By appointment (do not hesitate to do this!)

Why Am I Teaching This Class?

- Because I’m obsessed with things not working

Why Am I Teaching This Class?

Over time, I’ve wondered about:

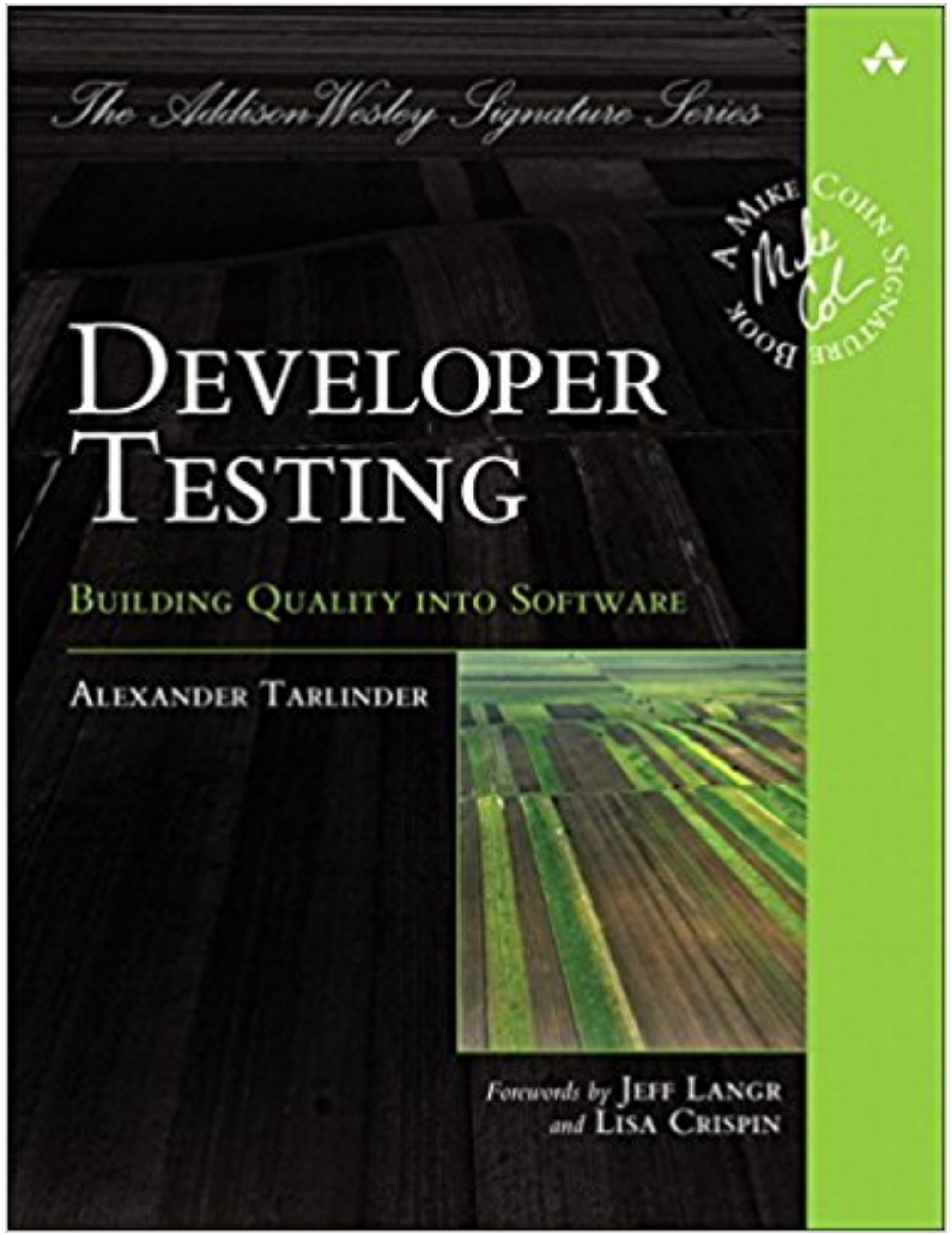
- “Why doesn’t my garbage collector work?”
- “Why doesn’t my OS kernel work?”
- “Why don’t (any) concurrent programs work?”
- “Why doesn’t my compiler work?”
- Now I wonder about lots of things, but mostly about general approaches to prevent or detect software defects
  - I’m back to thinking about OS kernels again. . .

Teaching Assistants

- We have 3 excellent TAs this term (shared across sections)
  - Enes Sezen
  - Ridwan Olawin
  - Srijan Pandey
- TA Hours will make use of:
  - Virtual CLC
  - Slack

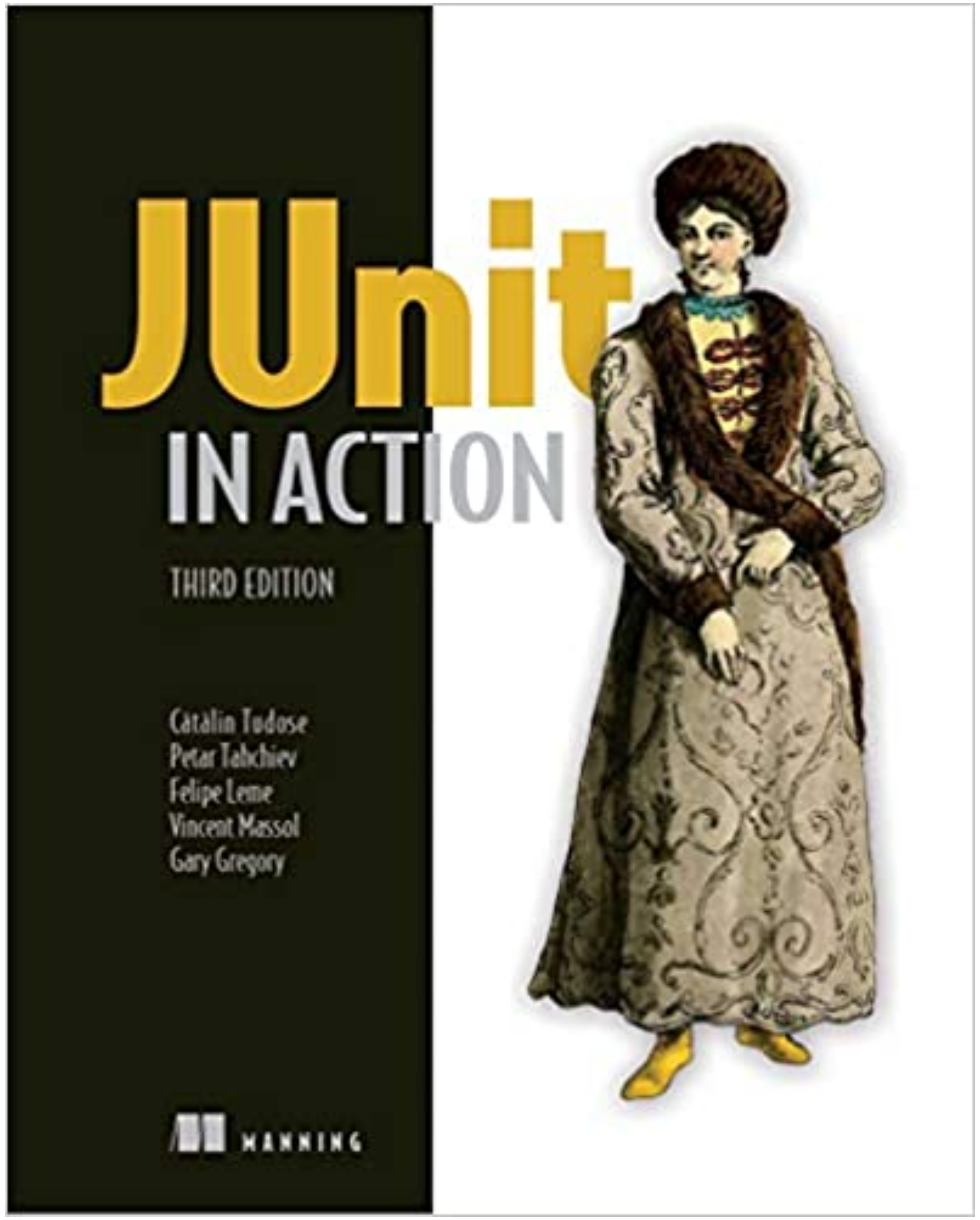
Textbook

- Developer Testing (2016), by Alexander Tarlinder
- Roughly half the course corresponds closely to groups of chapters
- The other half of the course is not well-covered by any generalist testing book
- Some material will only be in the lecture notes


The Addison-Wesley Signature Series
A Mike Cohn Signature Book
Developer Testing
Building Quality into Software
Alexander Tarlinder
Forewords by Jeff Langr and Lisa Crispin

Next Year's Textbook

- JUnit in Action, 3rd Edition, by Cătălin Tudose, Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory
- Better coverage of core techniques, including some topics current text doesn't cover
- Estimated December release :-/



Logistics

- Assignments handed in through BBLearn
  - Please make sure you hand in *all* files!
- Need help?
  - Office hours: TAs *or* professor
  - General questions about course material, assignments: Slack
    - Shared Slack with both sections
    - There’s an anonymity bot there if you’d like to be anonymous
    - Note: Faculty can see all messages, including DMs!
  - Other questions, like trouble specific to your code: don’t share in Slack!
    - Email *all of* the TAs and professor! This improves response time.

Lecture Logistics

- 3 hours is a long time to sit
- We will break around 80-90 minutes
- Questions during lecture?
  - Can ask via Zoom chat
  - Can ask via Zoom “hand raise”
  - Can just talk! “Excuse me, can you explain…”

Intended Audience

- This course is intended for *undergraduate students* in *Sofware Engineering* and *Computer Science*
- If you’re from another major (IS, Math, Game Design, etc.), welcome!
- Pre-requisite: CS260 Data Structures
- If you need to brush up on Java, *do so now*
  - Your first assignment goes out next week, though only uses basic Java
  - Every year someone who needs to do this, and knows it, doesn’t do it and sets themselves up for a rough term.
  - This includes Java generics.

Attendance

- The University requires attendance
- The world is a mess, so let’s be reasonable
- If you need to miss class, I’ll assume you’re mature enough to weigh the options responsibly. Just send me email so I know.
- Otherwise, I’ll track attendance via Zoom polls

Grading

- 6 Homeworks (80%)
  - Assignment 0 (10%): Basic Unit Testing
  - Assignment 1 (20%): Blackbox Unit Testing
  - Assignment 2 (20%): Whitebox Testing and Static Analysis
  - Assignment 3 (10%): Object-Oriented Testing
  - Assignment 4 (10%): Performance Testing
  - Assignment 5 (10%): GUI Testing
- Final Report (20%)

Term Grades

- Points-to-letter conversion in the syllabus
- I do not intend to curve the course, homework, or final

Grading Rules

- All grades are final
- There will be no extra credit assignments
- All late work will receive a reduced grade
  - -10% per week late
  - Maximum of 2 weeks late (after that, no credit)
  - Last two assignments are close to the end of the term, and therefore have reduced late periods
- If you hand in before the deadline, you may not hand in again after
- No extensions will be given beyond the end of the term
- No collaboraion is permitted during the exam, and assignments are *individual*

Extensions

- I generally prefer not to grant extensions
- But:
  - If you have a good reason (e.g., presenting at a conference or student research competition) and give sufficient notice, I'm open to extensions.
  - The tentative deadlines are on the syllabus right now.
  - If you ask for a last-minute extension for something you would have known about for a while, you'll not get an extension
  - Of course there are always emergencies.
- My view is that there are many good reasons an extension may be needed, and I don't want you to be penalized for unexpected life events. If you think you have a legitimate reason, ask! The worst I'll say is no.

Extensions

- To give you a little wiggle-room: you have one no-questions-asked 3 day extension on any homework assignment. You use it by simply informing me that you’re planning to use it.
  - Applies to any assignment, even at end of term, and the late policy starts from the end of the 3 days.

Academic Honesty

- The University’s academic honesty policy is in effect in this course. Please consult the student handbook for details.
- Higher order bit: Do not hand in work that is not your own, or not solely your own (modulo help from the professor and TA)
  - If you’re not sure if something is cheating, ASK FIRST!
  - You’re welcome to help each other *understand* assignments, but you shouldn’t be working out pseudocode together.
- Cheating is easier to catch than you think
  - Even *first-time* TAs catch it

If you cheat in this class, you will fail the class.

Cheating vs. Extensions

I recognize that most cheating is not mere laziness, but some combination of:

- Didn’t realize how much work it was, started too late.
- Close to the deadline, things are a mess, better to cheat and get the grade
- Viewing things as: grades are most important because they unlock future opportunities, and you can always learn stuff later

I recognize that these pressures exist, which is why there is a late policy, and a fairly flexible extension policy. My intent is for you to have enough legitimate flexibility that these shouldn’t motivate you to cheat; hence the strict penalty.

Malicious Cheating

Of course, some cheating *is* intentional.

A few previous years' students had their code public on Github.

- I have downloaded their code.
- I have emailed them to remove it.
- Github stats say some of those repositories had already been!

This year *all* student submissions will be run through MOSS alongside prior years' homeworks to find cheating.

- If you downloaded that code, best to just delete it.

Course Overview

What is This Course About?

Verification

How do we ensure software satisfies its requirements?

Validation

How do we ensure the software requirements satisfy the system’s intended use?

In other words…

Are we building the product correctly?
Are we building the correct product?

Software

A software product is composed of more than just code:

- Administrator manuals
- End-user guides
- Training materials
- Data (databases, audio, video, localization. . .)

When we talk about validating software, we really mean all of these things.

Software (cont.)

We’ll focus on just the software component: It’s the most technically challenging.

In the real world, these other components matter as much or more than the software!

- Can a non-technical user distinguish between incorrect documentation, unusable interfaces, and broken functionality?

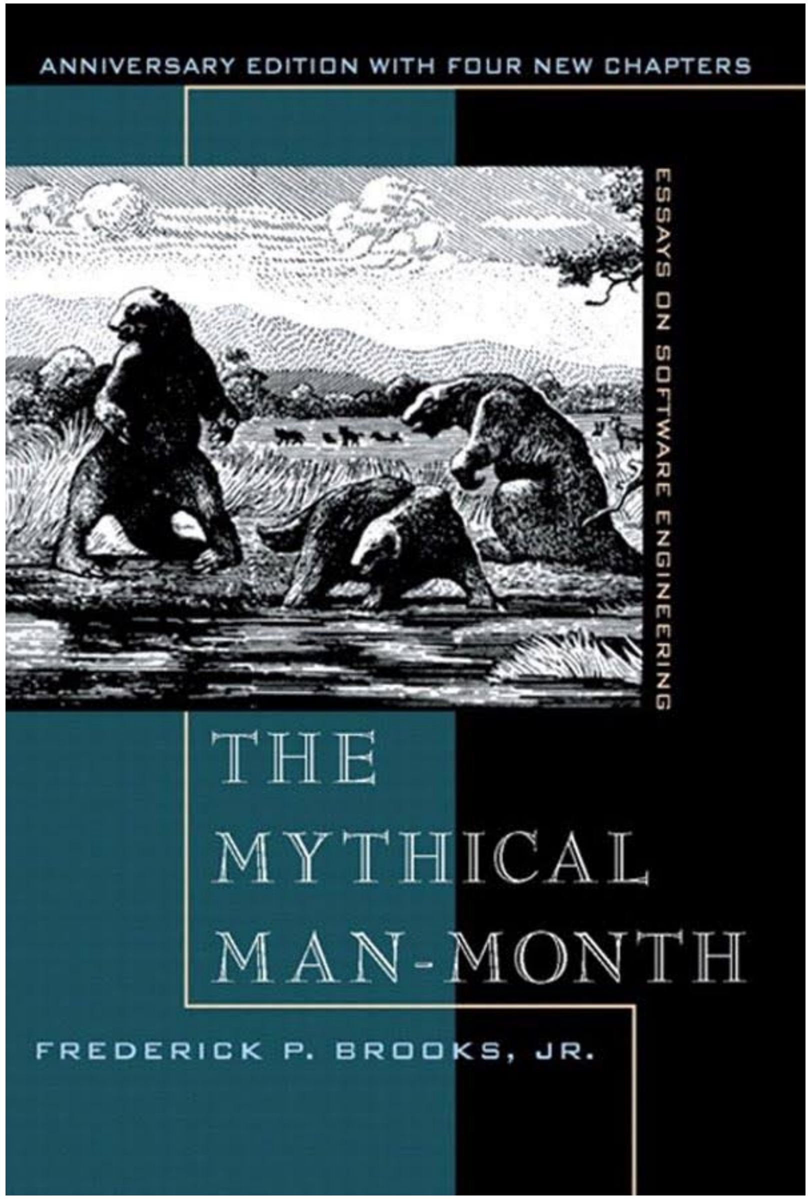
Who is “We”?

I’ve been speaking for some time now about things “we” can do. . . who is this “we”?

- It’s not the royal “we”
- It’s not the academic “we”
- It’s US! As developers and testers. . .

Personnel Roles in Software

- Historically, software development has been rigidly structured
- Separate roles for manager, architect, programmers, testers, …
- Increasingly not the case
  - *Especially* in smaller teams and startups!
- So what's a tester now?


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

You May Be A Tester If…

- Your job requirements include identifying software defects
- Your job requirements include producing *working code*
- Your job requirements include producing *secure code*
- Your job requirements include. . . code.

Today
This is most of a development team

An Aside on the Textbook

- I chose the textbook because it is the most cogent, modern take on testing I know of.
- But the textbook assumes there are distinct development and testing groups with different responsibilities.
  - This is no longer the case in many places!
- There is a new book coming out in December, Junit in Action (3rd Edition), from Manning Publications
  - Unfortunately, too late for this year, but better overall coverage

How Important Is Software Testing?

What do you think?

Does testing catch bugs that matter?

Medical Systems

Some of the most serious software failures have occurred in medical settings:

- The Therac-25 radiotherapy machine malfunctioned, causing massive overdoses of radiation to patients. (More in a moment)
- Pacemakers and several hundred other medical devices have been recalled due to faulty firmware/software
  - Recently, some have been recalled because they contained security flaws that would allow a malicious passer-by to e.g., shut off or overload a pacemaker. . .
- Medication records used for distributing medication throughout a hospital become inaccessible if e.g., the pharmacy database goes down. . .

Therac-25 Radiation Therapy

- In Texas, 1986, a man received between 16,500–25,000 rads in less than 1 second, over an area about 1 square centimeter
- He lost his left arm, and died of complications 5 months later
- In Texas, 1986, a man received 4,000 rads in the right temporal lobe of his brain
- The patient eventually died as a result of the overdose
- In Washington, 1987, a patient received 8,000-10,000 rads instead of the prescribed 86 rads.
- The patient died of complications of the radiation overdose.

Therac-25 (cont.)

The cause?

- Earlier hardware versions had a hardware interlock that shut off the machine if software requested a dangerous dose.
- Software on the earlier version never checked dosage safety; hardware checks masked the software bug
- Newer hardware removed the check (cost…)
- The software was not properly tested on the new hardware
  - Basis: it “worked” on the earlier hardware, which was almost the same
    - Other issues contributed as well
    - Nancy Leveson wrote the canonical report

Mars Climate Orbiter

- In 1999, NASA launched the Mars Climate Orbiter
- It cost $125 million (>184 million in 2017 USD)
- The spacecraft spent 286 days traveling to Mars
- Then it overshot...
- Lockheed Martin used English units
- NASA JPL used metric units
- The spec didn’t specify units, and nobody checked that the teams agreed.

Shaky Math

- In the US, 5 nuclear power plants were shut down in 1979 because of a program fault in a simulator program used to evaluate tolerance to earthquakes
- The program fault was found after the reactors were built!
- The bug? The arithmetic sum of a set of numbers was taken, instead of the sum of the absolute values.
- Result: The reactors would not have survived an earthquake of the same magnitude as the strongest recorded in the area.

AT&T Switch Boards

- In December 1989, AT&T installed new software in 114 electronic switching systems
- On January 15, 1990, 5 million calls were blocked during a 9 hour period nation wide
- The bug was traced to a C program that contained a `break` within a `switch` within a loop.
- Before the update, the code used `if-then-else` rather than `switch`, so the `break` exited the loop.
- After the conditions got too complex, a switch was introduced — and the break then only left the switch, not the loop!

Knight Capital

- On August 1, 2012, Knight Capital deployed untested code to their production high frequency trading servers.
- Well,7 out of 8
- The update reused an old setting that previously enabled some code to simulate market movements in testing
- When the “new” setting was enabled, it made the server with the old code act as if the markets were highly volatile
- The resulting trades lost the company $440 million immediately
- They barely stayed in business after recruiting new investors

Heartbleed

- Classic buffer overrun found in 2014
- OpenSSL accepted heartbeat requests that asked for too much data
- Server returned, e.g., private encryption keys
- Affected nearly every version of Linux (including Android) — most computers on the internet
  - Don’t worry, Mac got Shellshock a few months later
  - And shortly thereafter, Windows suffered similar bugs
- Now all major bugs come with logos and catchy names :-)

Ethereum “DAO Heist”

- Heard of cryptocurrency (e.g., Bitcoin?)
- Ethereum includes *smart contracts* — objects whose state and code is stored in the blockchain
- Accounts can expend small amounts to interact with smart contracts
- Smart contracts can manage ether (currency)
- Someone built an automated investment contract
- Someone else figured out how to withdraw more than they invested, and stole ~$150 million
- Cause: Allowing recursive calls to transfer *before* deducting from available client balance

fMRI Bugs

- Eklund et al. discovered the statistics software used in most fMRI studies and diagnoses was *never properly tested*
  - Eklund, Nichols, and Knutsson. Cluster Failure: Why fMRI Inferences for Spatial Extent have Inflated False-Positive Rates. PNAS July 2016.
- They found that errors in statistics packages (multiple) caused a high number of false positives.
- This questions *25 years* of fMRI research — over 40,000 studies! Not to mention patient treatments. . .

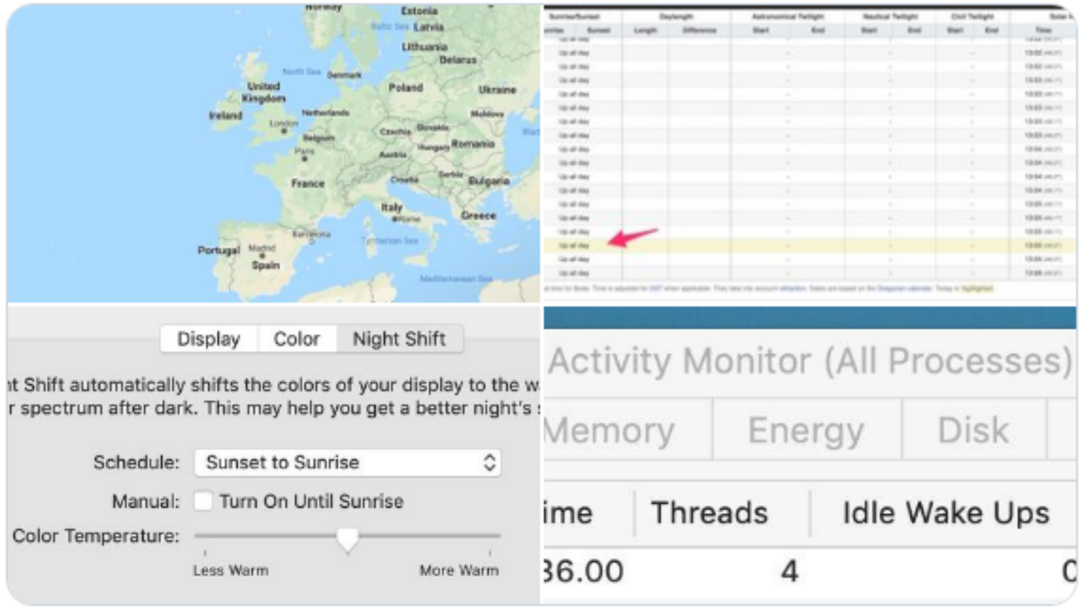
Boeing 737 MAX

- Two Boeing 737 MAX planes crashed in 2019
- Many things went wrong, but software was one aspect
- MCAS (Maneuvering Characteristics Augmentation System) responds to a sensor indicating plane’s angle
    - If the plane angles up too far (risking a stall), MCAS *automatically* tilts the horizontal tail to push the plane down
    - Doing this repeatedly is obviously bad, so there is a limit on how much MCAS can change this
    - BUT: software reset the limit tracking every time a pilot made *any* response, allowing MCAS to tilt the plane again
    - Combined with faulty sensors....
    - (There should have also been 2 sensors for redundancy, not 1)

Time Zones are Hard

Austin Johnsen
@AustinJ

TIL that if you go North of the Arctic Circle in the summer and bring a MacBook with Night Shift set to be triggered by sunrise/sunset, the process will go into an infinite loop because the sun never sets...



t Shift automatically shifts the colors of your display to the w
r spectrum after dark. This may help you get a better night's

Schedule: Sunset to Sunrise
Manual: ☐ Turn On Until Sunrise
Color Temperature: Less Warm — More Warm

Activity Monitor (All Processes)

ime	Threads	Idle Wake Ups
36.00	4	0

1:16 PM · Jun 28, 2019 · Twitter Web Client

Discussion…

Have you heard of other software bugs?

- In the media?
- From personal experience?

Does this embarrass you as a likely-future-software-engineer?

Defective Software

- We develop software that contains defects.
- It is likely the software we (including you!) will develop in the future will not be significantly better.

Back To Our Focus

What are things we — as testers — can do to ensure that the software we develop will satisfy its requirements, and when the user uses the software it will meet their actual needs?

Fundamental Factors in Software Quality

- Sound requirements
- Sound design
- Good programming practices
- Static analysis (code inspections, or via tools)
- Unit testing
- Integration testing
- System testing

Direct Impacts

Requirements & 3 major testing forms have *direct* impact

Sources of Problems

- Requirements Definition: Erroneous, incomplete, inconsistent requirements
- Design: Fundamental design flaws
- Implementation: Mistakes in programming, or bugs in dependencies
- Support Systems: Poor programming languages, faulty compilers and debuggers, misleading development tools
  - Did you know compilers and operating systems have bugs, too?
- Inadequate Testing of Software: Incomplete testing, poor verification, mistakes while debugging
- Evolution: Sloppy redevelopment or maintenance, introducing new flaws while fixing old flaws, incrementally

Requirements

- The quality of the requirements plays a critical role in the final product’s quality
- Remember *verification* and *validation*?
- Important questions to ask:
    - What do we know about the requirements’ quality?
    - What should we look for to make sure the requirements are good?
    - What can we do to improve the quality of the requirements?
- We’ll say a bit about requirements in this course. It’s covered more in SE181 / CS451

Specification

If you can’t say it, you can’t do it

You have to know what your product is before you can say whether it has a bug

Have you heard…?

It’s a feature, not a bug!

Specification

A specification defines the product being created, and includes:

- Functional Requirements that describe the features the product will support.
  - e.g., for a word processor, save, print, spell-check, font, etc. capabilities
- Non-functional Requirements that constrain how the product behaves
  - Security, reliability, usability, platform

Software Bugs Occur When…

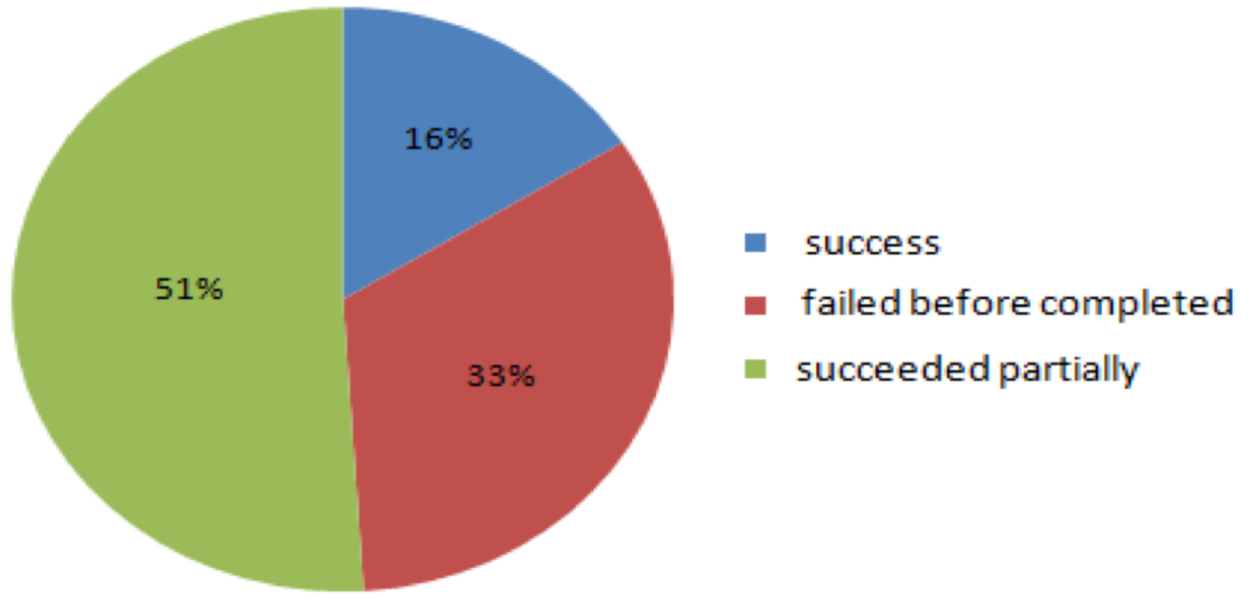
. . . at least one of these is true:

- The software does not do something that the specification says it should
- The software does something the specification says it should not do
- The software does not do something that the specification *does not* mention, *but should*
- The software is difficult to understand, hard to use, slow, . . .

Many Bugs are Not From Coding Errors!

- Wrong specification?
  - No way to write correct code
- Poor design?
  - Good luck debugging
- Bad assumptions about your platform (OS), threat model, network speed. . .

The Requirements Problem: Standish Report (1995)



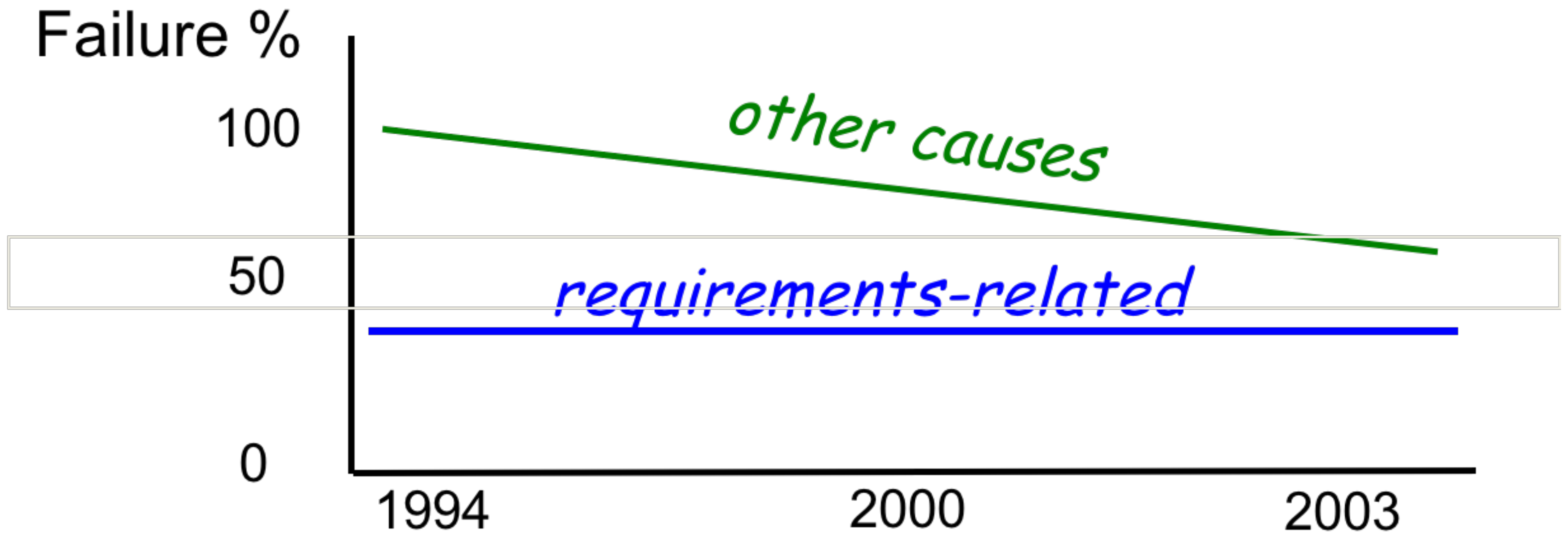
Major Source of Failure

Poor requirements engineering: roughly 50% of responses.

The Requirements Problem: European Survey (1996)

- Coverage: 3800 European organizations, 17 countries
- Main software problems perceived to be in
  - Requirements Specification: > 50%
  - Requirements Evolution Management: 50%

The Requirements Problem Persists…



J. Maresco, IBM developerWorks, 2007

Relative Cost of Bugs

- Cost to fix a bug increases exponentially (10^*t*)
  - i.e., it increases tenfold as time increases
- E.g., a bug found during specification costs $1 to fix
- ... if found in design it costs $10 to fix
- ... if found in coding it costs $100 to fix
- … if found in released software it costs $1000 to fix

These are rule-of-thumb: different studies find different ratios, but the costs always increase.

Bug Free Software

Software is in the news for all the wrong reasons

- Security breaches, hackers getting credit card information, hacked political emails, etc.

Why can’t developers just write software that works?

- As software gets more features and supports more platforms, it becomes increasingly difficult to make it bug-free.

Discussion

- Do you think bug free software is unattainable?
  - Are there technical barriers that make this impossible?
  - Is it just a question of time before we can do this?
  - Are we missing technology or processes?

Formal Verification

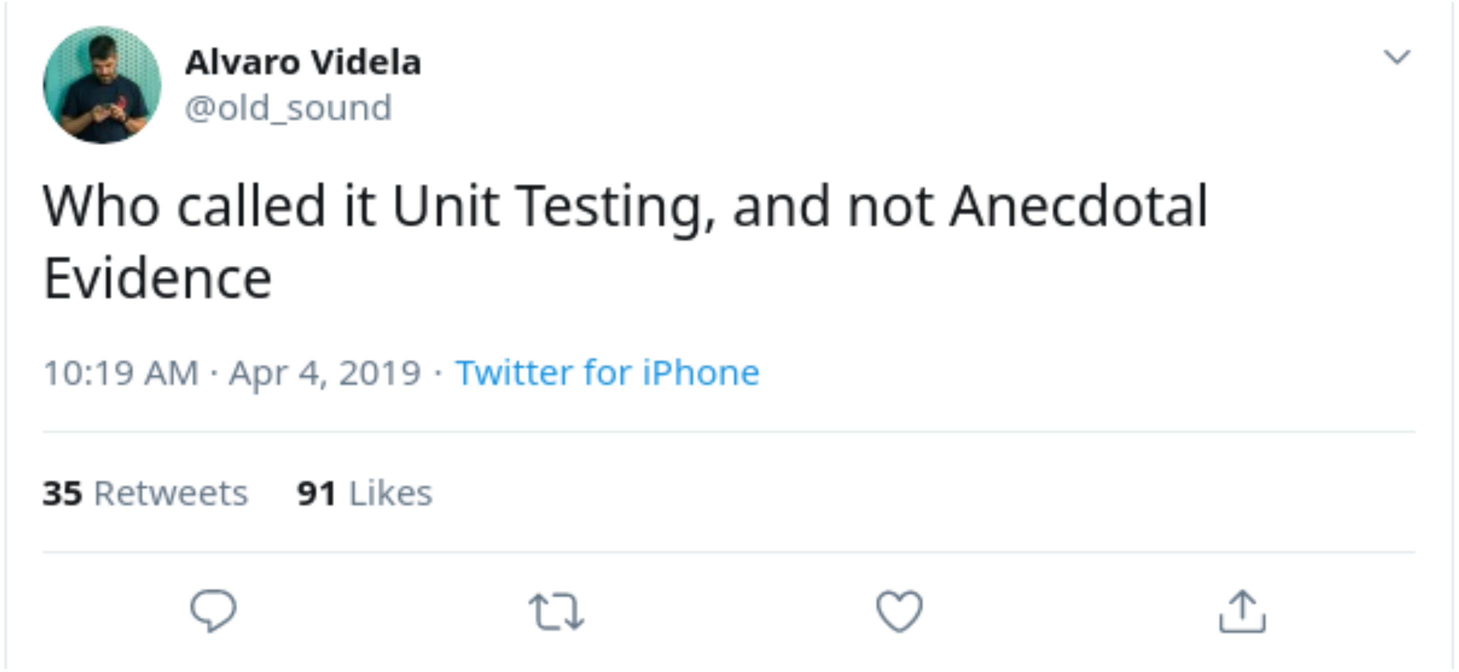
- Use lots of math to prove properties about programs!
  - Lots of math, but aided by computer reasoning
- The good:
  - It can in principle eliminate any class of bugs you care to specify
  - It works on real systems now (OS, compiler, distributed systems)
- The bad:
  - Requires far more time/expertise than most have
  - Verification tools are still software
  - Verified software is only as good as your spec!
  - Still not a good financial decision for *most* software
  - Exceptions: safety-critical, reusable infrastructure

So, What Are We Doing?

- In general, it’s not yet practical to *prove* software correct
- So what do we do instead?
- We collect evidence that software is correct
  - Behavior on representative/important inputs (tests)
  - Behavior under load (stress/performance testing)
  - Stare really hard (code review)
  - Run lightweight analysis tools without formal guarantees, but which are effective at finding issues

Evidence

Why is this okay?


Alvaro Videla
@old_sound

Who called it Unit Testing, and not Anecdotal Evidence

10:19 AM · Apr 4, 2019 · Twitter for iPhone

35 Retweets **91** Likes

We can do this in a *principled way* that allows us to gather *strong* evidence.

Goals of a Software Tester

- To find bugs
- To find them as early as possible
- To make sure they get fixed

Note

Doesn’t say *eliminate all bugs*. This would be wildly unrealistic for the forseeable future.

The Software Development Process

Discussion

- What is software engineering?
- Where/when does testing occur in the software development process?

Software is…

- requirements specification documents
- design documents
- source code
- test suites and test plans
- interfaces to hardware and software operating environments
- internal and external documentation
- executable programs and their persistent data

Software Effort is Spent On…

- Specification
- Product reviews
- Design
- Scheduling
- Feedback
- Competitive information acquisition
- Test planning
- Customer surveys
- Usability data gathering
- Look and feel specification
- Software architecture
- Programming
- Testing
- Debugging

Software Effort is Spent On…

- Project managers
    - Write speciifcation, manage the schedule, make critical decisions about trade-offs
- Software architects, system engineers
    - Design & architecture, work closely with developers
- Programmers/developers/coders
    - Write code, fix bugs
- Testers, quality assurance (QA)
    - Find bugs, document bugs, track progress on open bugs
- Technical writers
    - Write manuals, online documentation
- Configuration managers, builders
    - Packaging and code, documents, specifications

Software Project Staff Include…

Note

Today people usually hold multiple roles!

Architect, Programmer, and Tester are increasingly merged into a single role, though the balance changes as seniority increases.

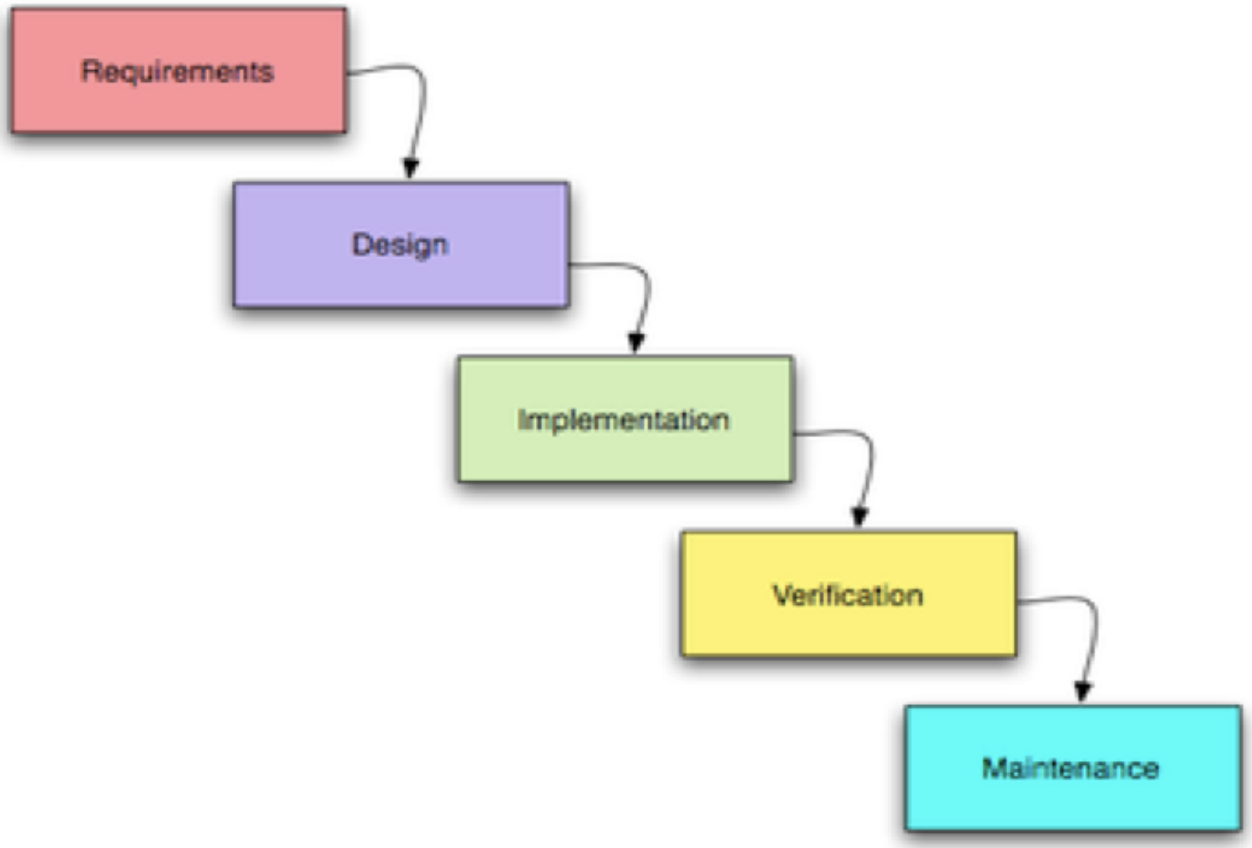
Development vs. Testing

- Many sources, including the textbook, make a strong distinction between development and testing
- *I do not.*
- Development and testing are two highly complementary and largely overlapping skill sets and areas of expertise — it is not useful to draw a clear line between the two
- Historically, testers and developers were disjoint teams that rarely spoke
    - We'll talk about some of the dysfunction this caused
- Today, many companies have only one job title, within which one can specialize towards new development or testing
- For our purposes, a tester is anyone who is responsible for code quality.

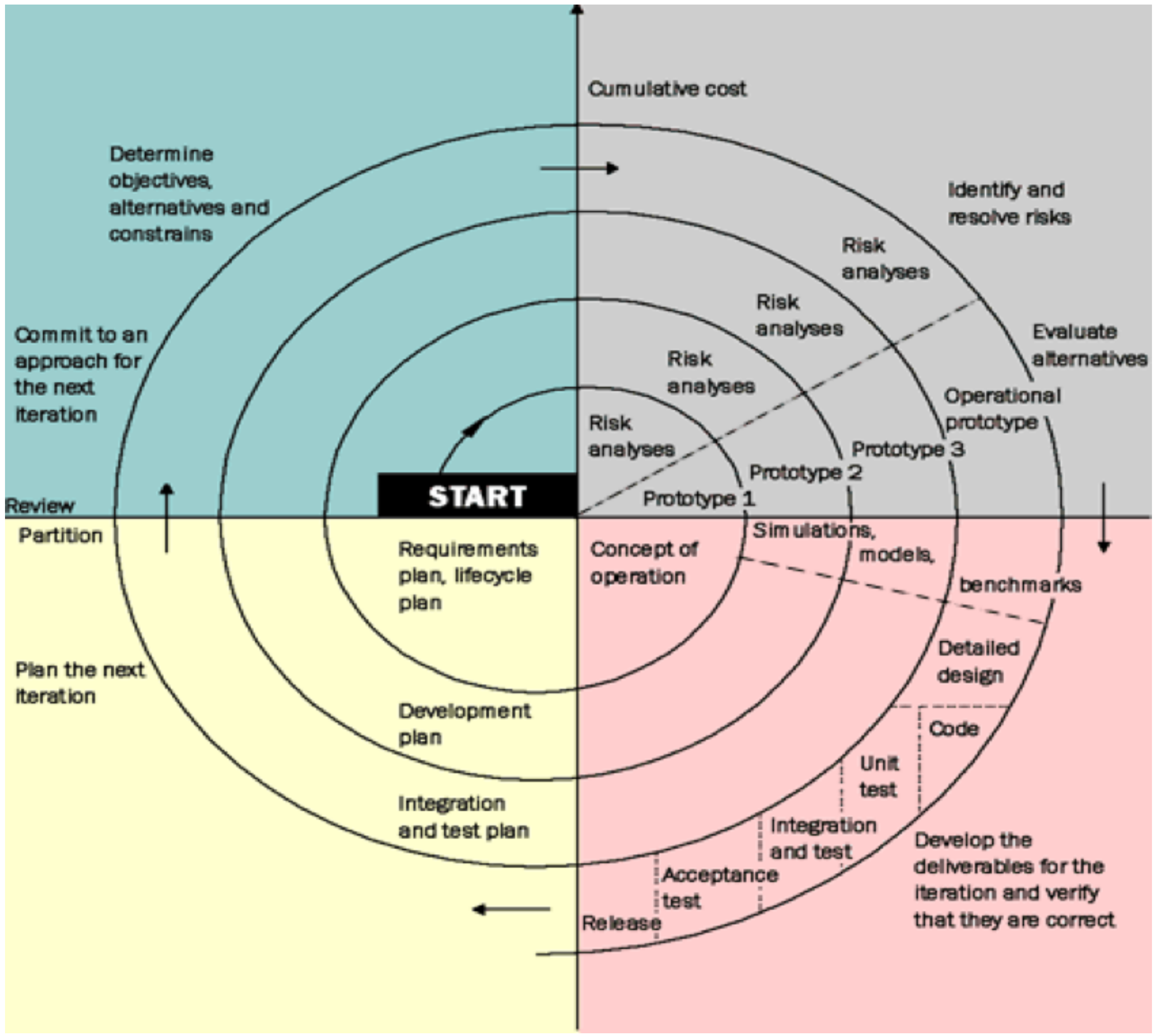
Development Styles

- Code and Fix
- Waterfall
- Spiral
- Agile
  - Scrum
  - XP
  - Test-Driven Development
  - Behavior-Driven Development

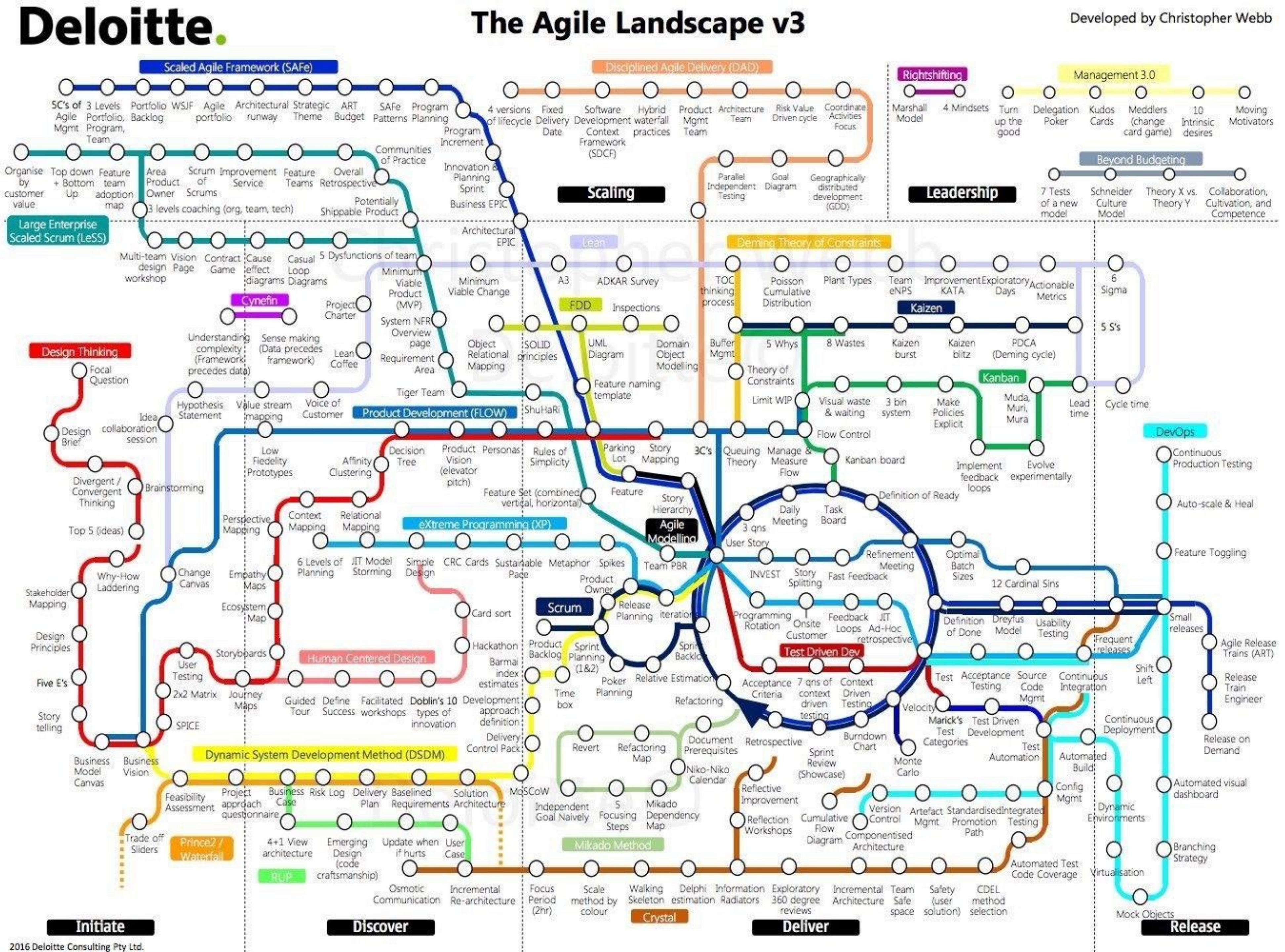
Waterfall


Requirements
Design
Implementation
Verification
Maintenance

Spiral


Cumulative cost
Determine objectives, alternatives and constrains
Identify and resolve risks
Risk analyses
Risk analyses
Risk analyses
Risk analyses
Evaluate alternatives
Operational prototype
Prototype 3
Prototype 2
Prototype 1
Commit to an approach for the next iteration
START
Review
Partition
Requirements plan, lifecycle plan
Concept of operation
Simulations, models, benchmarks
Detailed design
Code
Unit test
Integration and test
Acceptance test
Release
Plan the next iteration
Development plan
Integration and test plan
Develop the deliverables for the iteration and verify that they are correct

Corporate Agile



A Grain of Salt

- Between Waterfall, Sprial, Agile, XP, Scrum, TDD, BDD, and dozens of other approaches:
  - Everyone says they do X
  - Few do exactly X
  - Most borrow main ideas from X and a few others, then adapt as needed to their team, environment, or other influences
  - But knowing the details of X is still important for communication, planning, and understanding trade-offs
- Key element of success: adaptability
  - The approaches that work well tend to assume bugs and requirements changes will occur, so plan on revisiting old code

The Original Waterfall Picture

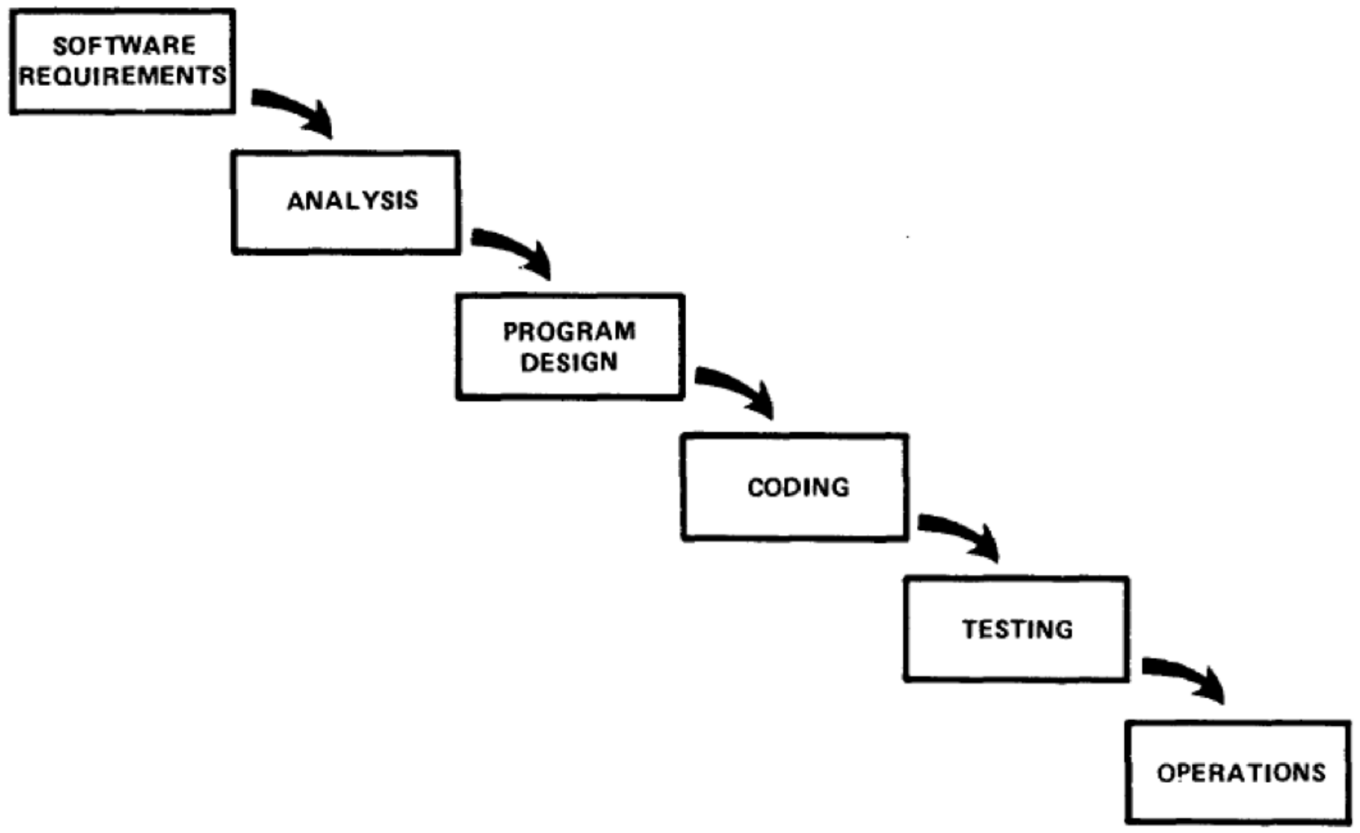
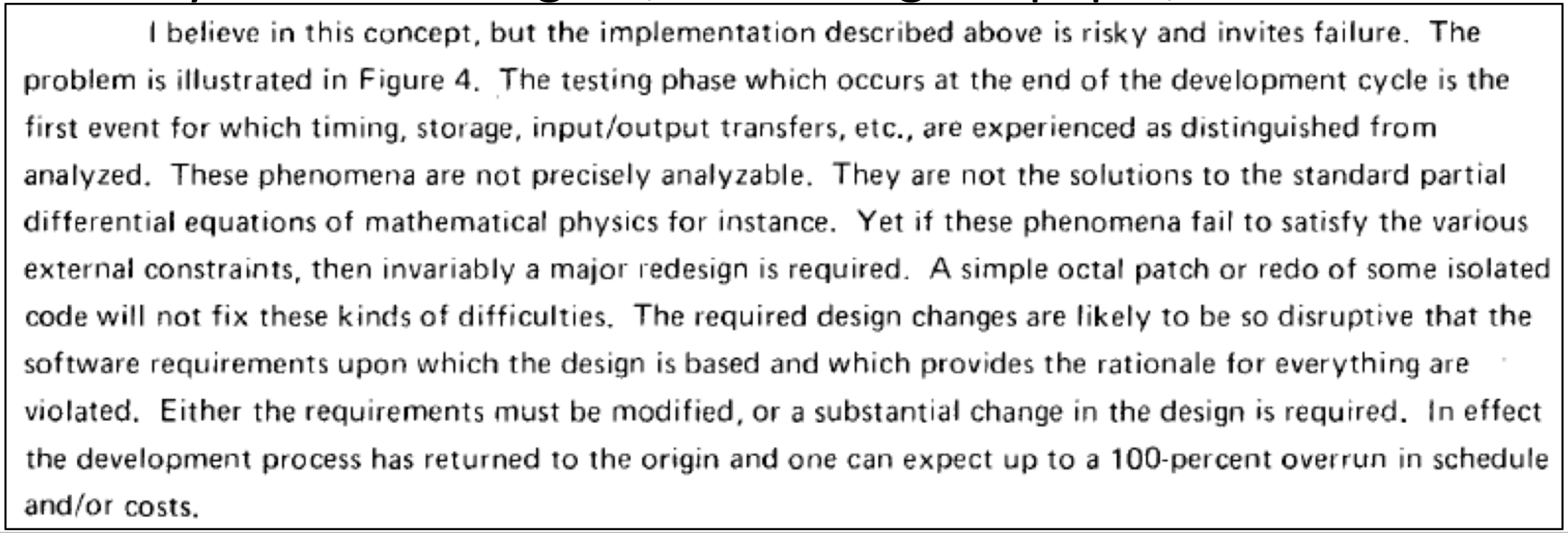


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

Royce, W. Managing the Development of Large Software Systems. IEEE WESCON, 1970.

Describing the Original Waterfall Diagram

Immediately below that figure, in the original paper, is this:



I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4. The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.

Key Sentence

I believe in this concept, but the implementation described above is risky and invites failure.

Waterfall Improved

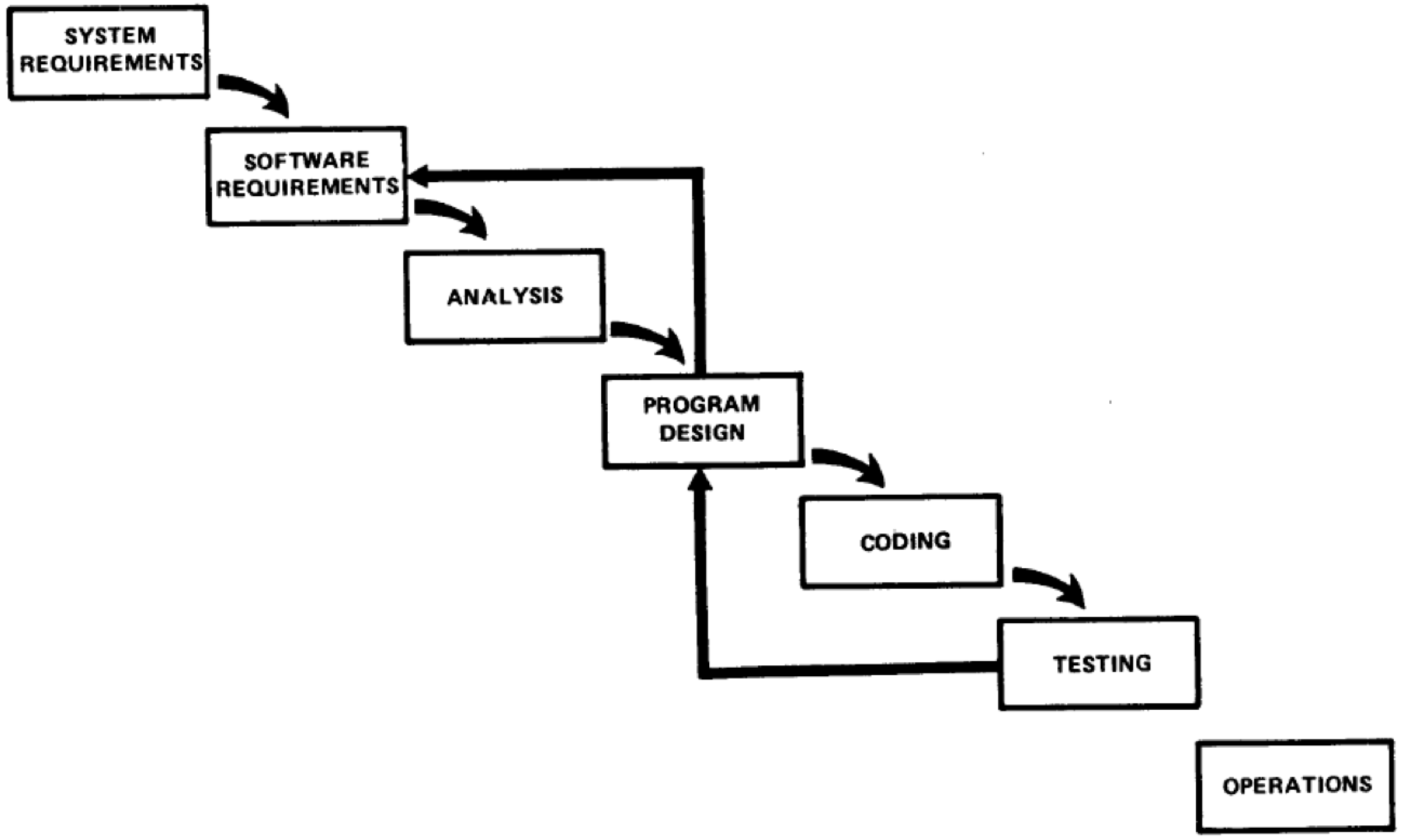


Figure 4. Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.

Two Styles of Testing

Traditional Testing (Waterfall, etc.)

- Verification after construction
- Assumes clear specification exists ahead of time
- Assumes developers and testers interpret the spec the same way…

Agile Testing (TDD, BDD, Scrum, etc.)

- Testing throughout development
- Developers and testers collaborate, iterate together rapidly
- Assumes course-corrections will be required frequently
- Emphasizes feedback and adaptability

Two Philosophies for Testing

Testing to Critique

- Does the software meet its specification?
- Is it usable? Is it fast enough?
- Does this comply with relevant legal requirements?
- Emphasis on testing completed components

Testing to Support

- Does what we’ve implemented so far form a solid basis for further development?
- Is the software so far reliable?
- Emphasis on iterative feedback *during development*

Testing Vocabulary

An Overview of Testing

- We’ve already mentioned many types of testing in passing
  - Unit tests
  - Integration tests
  - System tests
  - Usability tests
  - Performance tests
  - Functional tests
  - Nonfunctional tests
- What do these (and more) all mean?
- How do they fit together?
- To talk about these, we need to set out some terminology

Errors, Defects, and Failures

- Many software engineers use the following language to distinguish related parts of software issues:
  - An error is a mistake made by the developer, leading them to produce incorrect code
  - A defect is the problem in the code.
    - This is what we commonly call a bug.
  - A failure occurs when a defect/bug leads the software to exhibit incorrect behavior
    - Crashes
    - Wrong output
    - Leaking private information

Errors, Defects, and Failures (cont.)

- Not every defect leads to a failure!
    - Some silently corrupt data for weeks and months, and *maybe* eventually cause a failure
    - Some teams use a distinct term for when a mistake leads to incorrect *internal* behavior, separately from *external* behavior
- Some failures are not caused by defects!
    - If you hold an incandescent light bulb next to a CPU, random bits start flipping. . . .

Forcing Failures with a Light Bulb

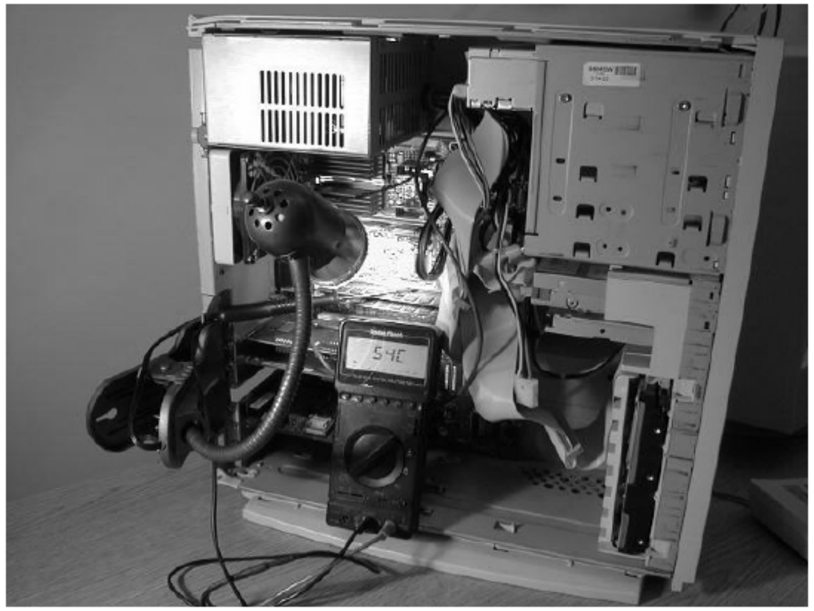


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

From Govindavajhala & Appel's IEEE Security and Privacy 2003 paper, *Using Memory Errors to Attack a Virtual Machine*.

Alternative Language

- This error/defect/failure terminology is not universal
  - It is *common*
- What terminology you use isn’t really important, as long as your team agrees
  - The point of this terminology isn’t pedantry
  - The point of this terminology is communication, which is more important than particular terms
- In this course, we’ll stick to error/defect/failure

Open-box and Closed-box Testing

Open Box Testing (aka, Whitebox)

- Testing software *with knowledge of its internals*
- A developer-centric perspective
- Testing implementation details

Closed Box Testing (aka Blackbox)

- Testing software *without* knowledge of its internals
- A user-centric (external) perspective
- Testing external interface contract

These are *complementary*; we'll discuss more later.

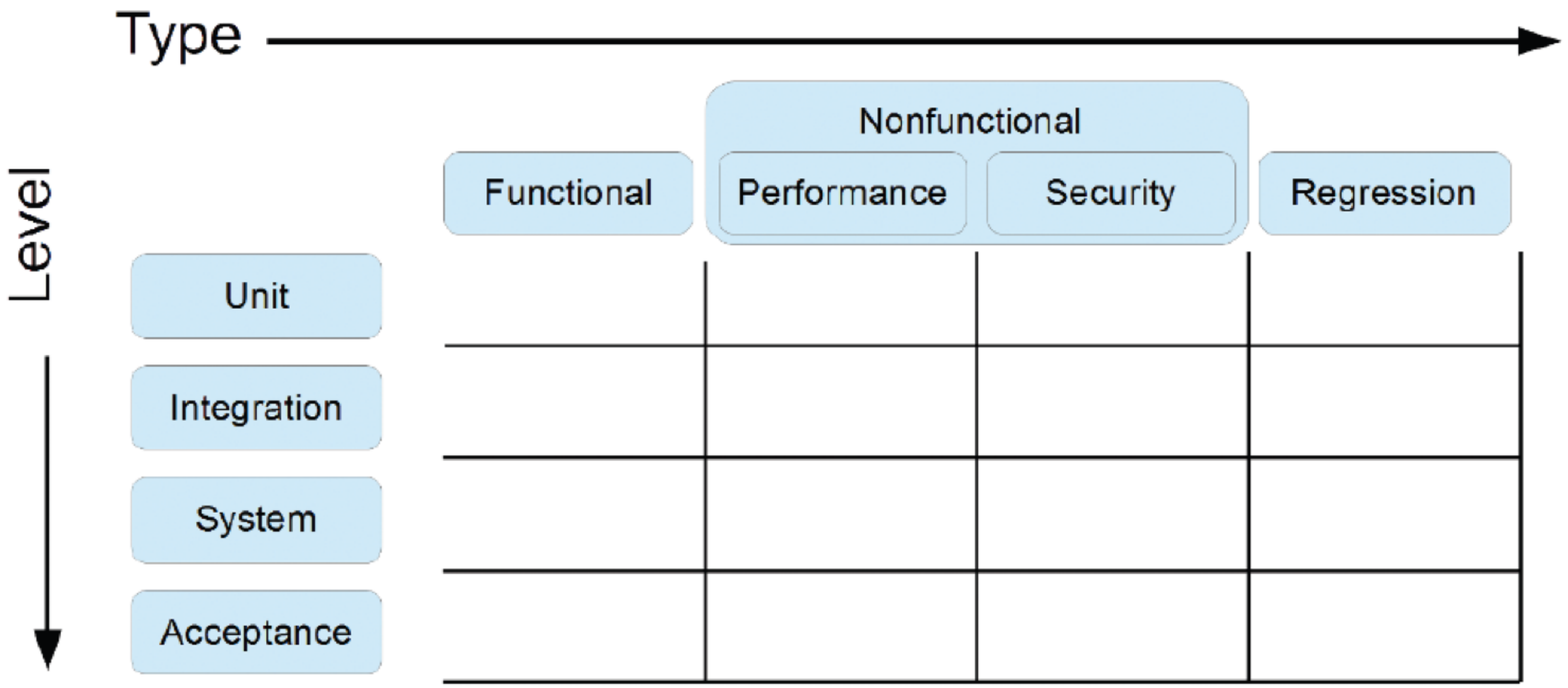
Classifying Tests

There are two primary “axes” by which tests can be categorized:

- Test Levels describes the “level of detail” for a test: small implementation units, combining subsystems, complete product tests, or client-based tests for accepting delivery of software
- Test Types describe the goal for a particular test: to check functionality, performance, security, etc.

Each combination of these can be done via critiquing or support, in closed box or open box fashion.

Classifying Tests



Why Classify?

Before we get into the details of that table, why even care?

Having a systematic breakdown of the testing space helps:

- Planning — it provides a list of what needs to happen
  - Different types of tests require different infrastructure
  - Determines what tests can be run on developer machines, on every commit, nightly, weekly, etc.
- Division of labor
  - Different team members might be better at different types of testing

Why Classify? (cont.)

- Exposes the option to skip some testing
  - Never ideal, but under time constraints it provides a menu of options
- Checking
  - Can’t be sure at the end you’ve done all the testing you wanted, if you didn’t know the options to start!

Test Levels

Four standard levels of detail:

- Unit
- Integration
- System
- Acceptance

Have you heard of these before?

Unit Tests

- Testing smallest “units of functionality”
- Intended to be fast (quick to run a single test)
  - Goal is to run all unit tests frequently (e.g., every commit)
- Run by a unit testing framework
- Consistently written by developers, even when dedicated testers exist
- Typically open box, but not always
  - Any unit test of internal interface is open box
  - Testing external APIs can be closed box

Units of Functionality

Unit tests target small “units of functionality.” What’s that?

- Is it a method? A class?
- What if the method/class depends on other methods/classes?
  - Do we “stub them out” (more later)
  - Do we just let them run?
- What if a well defined piece of functionality depends on multiple classes?

There is no single right answer to these questions.

Guidelines for Unit Tests

- Well-defined single piece of functionality
- Functionality independent of environment
- Can be checked independently of other functionality
    - i.e., if the test fails, you know precisely which functionality is broken

Examples of “Definite” Unit Tests

- Insert an element into a data structure, check that it’s present
- Pass invalid input to a method, check the error code or exception is appropriate
  - Specific way the input is invalid: out of bounds, wrong state, …
- Sort a collection, check that it’s sorted

Gray Areas

Larger pieces of functionality can still be unit tests, but may be integration. Unfortunately, some “I know it when I see it”

Concrete Unit Test

```
@Test
public void testMin02() {
 int a = 0, b = 2;
 int m = min(a,b);
 assertSame("min(0,2)␣is␣0", 0, m);
}
```

Challenges for Unit Tests

- Size and scope (as discussed)
- Speed
- How do you know you have enough?
  - More on this with openbox testing / code coverage
- Might need stubs
  - Might need to “mock ups” of expensive resources like disks, databases, network
  - Might need a way to test control logic without physical side effects

System Tests

- Testing *overall* system functionality, for a complete system
- Assumes all components already work well
- Reconciles software against top-level requirements
  - Tests stem from concrete use cases in the requirements

But wait — we skipped a level!

Integration Tests

- A test checking that two “components” of a system work together
  - Yes, this is vague
- Emphasizes checking that components implement their *interfaces* correctly
  - Not just Java interfaces, but the expected behavior of the component
- Testing combination of components that are larger than unit test targets
  - Not testing the full system
- Many tests end up as integration tests by process of elimination — not a unit test, not a system test, and therefore an integration test.

Two Approaches to Integration

Big Bang

- Build everything.
- Test individually.
- Put it all together.
- Do system Tests

Incremental

- Test individual components
- Then pairs
- Then threes…
- Until you finish the system.

Big Bang Integration

Advantages:

- Everything is available to test

Disadvantages:

- All components might not be ready at the same time
- Focus is not on a specific component
- Hard to locate errors
    - Which component is at fault for a failed test?

Incremental Integration

Advantages:

- Focus is on individual or smaller sets of modules at a time
- Easier to track down sources of problems

Disadvantages:

- Need to develop special code (stubs and/or drivers)
  - Leads to top-down or bottom-up integration
  - Caveat: You’ll need these anyways, just possibly less

Acceptance Tests

- Performed by a customer / client / end user
- Testing to see if the customer believes the software is what they wanted
- Also tied to requirements and use cases

Test Types

Test “types” classify the purpose of the test, rather than its scope or mechanism. Let’s talk about:

- Functional testing
- Non-functional testing
- Performance testing
- Security testing
- Regression testing

Note these aren’t all mutually exclusive!

Functional Testing

- The default assumption in testing
- Functional testing verifies the software’s behavior matches expectations.
- Also includes testing bad inputs, to check implicit assumptions
  - i.e., given nonsense input, the program should do “something reasonable”
  - i.e., the compiler shouldn’t delete your code if you have a type error
- Cuts across all levels of testing, but heavy on unit testing

Nonfunctional Testing

- Roughly, testing things that are not “functionality”
- Generally, tests quality of the software, also called the “-ilities”
    - Usability
    - Reliability
    - Maintainability
    - Security
    - Performance

Nonfunctional testing is an umbrella term for many test types.

Functional vs. Nonfunctional

Functional testing concerns *what* the software does.
Nonfunctional testing concerns *how* it does it.

Performance Testing (Nonfunctional)

Performance testing is, broadly, how quickly the software works. But it includes a variety of subtypes:

- Performance Testing without further qualification test how fast the software performs certain tasks
- Load Testing checks how the system performs with a high number of users
- Stress Testing checks how the system handles having more users/requests than it was designed for
- Spike Testing check how the system handles high stress that arrives *suddenly*

This is a complex area. We'll spend a lecture on the absolute basics later this term, but it's possible to run a whole course on this.

Security Testing (Nonfunctional)

Security testing ensures the system is *secure*, which has a more nuanced meaning than most assume. A common model of “secure” is the “CIA security triad:”

- **C**onfidentiality
  - Data confidentiality (private information stays private)
  - Privacy (control over private data)
- **I**ntegrity
  - Data integrity (reliable data storage)
  - System integrity (system cannot be compromised/hacked)
- **A**vailability
  - Resources are available to authorized users and no one else

Another topic that could fill a whole course (or several).

Regression Testing

Making sure that code changes haven’t broken existing functionality, performance, security, etc.

The Need for Regression Testing

It’s common to introduce new bugs while changing existing code, whether fixing an earlier bug or adding a new feature.

- In practice, this means re-running tests after a code change
- With good test automation and good unit/integration/system/etc. tests, this is literally running tests again after a change.
- Next week we’ll talk about continuous integration, which directly addresses this

Testing Costs

We haven’t discussed the *cost* of tests — only a bit about their logistics and purposes.

- Ideally we’d write tests for every conceivable thing, and re-run every test on every change.
- Then we know immediately whether functionality was broken
- But nobody does this — why?

Testing Costs

- In general, there are *always more bugs*, but we can't write tests forever.
- Must prioritize likely scenarios (common use patterns) and high-risk scenarios (e.g., security)
- Some exceedingly rare cases may not be tested! Maybe in V2...
- For large systems, running all tests takes too long.
    - Running all tests for Microsoft Windows, end to end, on one
    - machine, would take months.
    - This is infeasible to do for every change.
    - A subset of fast tests (e.g., unit tests) is run on every change.
    - Other tests are run nightly or weekly depending on cost.